The Ciao CLP($\mathcal{FD}$) Library
A Modular CLP Extension for Prolog
(System Description)

Emilio Jesús Gallego Arias[1], Rémy Haemmerlé[1],
Manuel V. Hermenegildo[1,2], and José F. Morales[2]

[1] Universidad Politécnica de Madrid
[2] IMDEA Software Institute

Abstract. We present a new free library for Constraint Logic Programming over Finite Domains, included with the Ciao Prolog system. The library is entirely written in Prolog, leveraging on Ciao's module system and code transformation capabilities in order to achieve a highly modular design without compromising performance. We describe the interface, implementation, and design rationale of each modular component. The library meets several design goals: a high level of modularity, allowing the individual components to be replaced by different versions; high-efficiency, being competitive with other $\mathcal{FD}$ implementations; a glass-box approach, so the user can specify new constraints at different levels; and a Prolog implementation, in order to ease the integration with Ciao's code analysis components. The core is built upon two small libraries which implement integer ranges and closures. On top of that, a *finite domain variable* datatype is defined, taking care of constraint reexecution depending on range changes. These three libraries form what we call the *$\mathcal{FD}$ kernel* of the library. This $\mathcal{FD}$ kernel is used in turn to implement several higher-level finite domain constraints, specified using indexicals. Together with a labeling module this layer forms what we name *the $\mathcal{FD}$ solver*. A final level integrates the CLP($\mathcal{FD}$) paradigm with our $\mathcal{FD}$ solver. This is achieved using attributed variables and a compiler from the CLP($\mathcal{FD}$) language to the set of constraints provided by the solver. It should be noted that the user of the library is encouraged to work in any of those levels as seen convenient: from writing a new range module to enriching the set of $\mathcal{FD}$ constraints by writing new indexicals.

1 Introduction

Constraint Logic Programming (CLP) [1] is a natural and well understood extension of Logic Programming (LP) in which term unification is replaced by constraint solving over a specific domain. This brings a number of theoretical and practical advantages which include increased expressive power and declarativeness, as well as higher performance for certain application domains. The resulting CLP languages allow applying efficient, incremental constraint solving techniques to a variety of problems in a very natural way: constraint solving

blends in elegantly with the search facilities and the ability to represent partially determined data that are inherent to logic programming. As a result, many modern Prolog systems offer different constraint solving capabilities.

One of the most successful instances of CLP is the class of constraint logic languages using *Finite Domains* ($\mathcal{FD}$). Finite domains refer to those constraint systems in which constraint variables can take values out of a finite set, typically of integers (i.e., a *range*). They are very useful in a wide variety of problems, and thus many Prolog systems offering constraint solving capabilities include a finite domain solver. In such systems, domain (range) definition constraints as well as integer arithmetic and comparison constraints are provided in order to specify problems.

Since the seminal paper of Van Hentenryck et al. [2], many FD solvers adopt the so-called "glass-box" approach. Our FD Kernel also follows this approach, based on a unique primitive called an *indexical*. High-level constraints are then built/defined in terms of primitive constraints. An indexical has the form `X in r`, where r is a range expression (defined in **??**). Intuitively, `X in r` constrains the $\mathcal{FD}$ term ($\mathcal{FD}$ variable or integer) `X` to belong to the range denoted by the term `r`. In the definition of the range special expressions are allowed. In particular, the expressions `max(Y)` and `max(Y)` evaluate to the minimum and the maximum of the range of the $\mathcal{FD}$ variable `Y`, and the expression `dom(Y)` evaluates to the current domain of `Y`. Constrains are solved partially in an incremental using consistency techniques [3] which maintain the constraint network in some coherent state (depending on the arc-consistency algorithm used). This is done by monotone domain shrinking and propagation. When all constraints are placed and all values have been propagated a call is typically made to a *labeling* predicate which performs an enumeration-based search for sets of compatible instantiations for each of the variables that remain not bound to a single value. We refer to [2] for more details regarding indexicals and finite domain constraint solving.

In this paper, we present a new free library for Constraint Logic Programming over Finite Domains, included with the Ciao Prolog system [4]. The library is entirely written in Prolog, leveraging on Ciao's module system and code transformation capabilities in order to achieve a highly modular design without compromising performance. We describe the interface, implementation, and design rationale of each modular component. The library meets several design goals: a high level of modularity, allowing the individual components to be replaced by different versions; high-efficiency, being competitive with other $\mathcal{FD}$ implementations; a glass-box approach, so the user can specify new constraints at different levels; and a Prolog implementation, in order to ease the integration with Ciao's code analysis components. The core is built upon two small libraries which implement integer ranges and closures. On top of that, a *finite domain variable* datatype is defined, taking care of constraint reexecution depending on range changes. These three libraries form what we call the $\mathcal{FD}$ *kernel* of the library. This $\mathcal{FD}$ kernel is used in turn to implement several higher-level finite domain constraints, specified using indexicals. Together with a labeling module this layer forms what we name the $\mathcal{FD}$ *solver*. A final level integrates

the CLP($\mathcal{FD}$) paradigm with our $\mathcal{FD}$ solver. This is achieved using attributed variables and a compiler from the CLP($\mathcal{FD}$) language to the set of constraints provided by the solver. It should be noted that the user of the library is encouraged to work in any of those levels as seen convenient: from writing a new range module to enriching the set of $\mathcal{FD}$ constraints by writing new indexicals.

One of the first CLP($\mathcal{FD}$) implementations is the CHIP system [5]. This commercial system follows a typical black-box approach: it consists of a complete solver written in C and interfaces in an opaque manner to a Prolog engine. This makes it difficult for the programmer to understand what is happening in the core of the system. Also, no facilities are provided for tweaking the solver algorithms for a specific application.

More recent CLP($\mathcal{FD}$) systems such as those in SICStus [6], GNU Prolog [7,8], and B-Prolog [9] are built instead following more the glass-box approach. The basic constraints are decomposed into smaller but highly optimized primitives (typically indexicals). Consequently, the programmer has more latitude to extend the constraints as needed. However, even if such systems can be easily modified/extended at the interface level (e.g., both SICStus and B-Prolog provide way to define new global constraints) they are much harder to modify at the implementation level (e.g., it is not possible to replace the implementation of range).

The Ciao CLP($\mathcal{FD}$) library that we present has more similarities with the one recently developed for SWI Prolog [10]. Both are fully written in Prolog and support unbound ranges. The SWI library is clearly more complete than Ciao's (e.g., it provides some global constraints and always terminating propagation), but it is designed in a monolithic way: it is implemented in a single file, mixing different language extensions (using classical Prolog `term_expansion` mechanisms) while the Ciao library is split in more around 20 modules with a clear separation of the different language extensions [11].

Summarizing, our library differs in a number of ways from other existing approaches:

- First, along with more recent libraries it differs from early systems in that it is written entirely in Prolog. This dispenses with the need for a foreign interface and opens up more opportunities for automatic program transformation and analysis. The use of the meta-predicates `setarg/3` and `call/1` means that the use of Prolog has a minimal impact on performance.
- Second, the library is designed as a set of separate modules. This allows replacing a performance-critical part — like the range code — with a new implementation better suited for it.
- Third, the library supports the "glass-box" approach fully, encouraging the user to access directly the low-level layers for performance-critical code without losing the convenience of the high-level CLP paradigm. Again, the fact that the implementation is fully in Prolog is the main enabler of this feature.
- Lastly, we have prioritized extensibility, ease of modification, and flexibility, rather than micro-optimizations and pure raw speed. However, we argue that

our design will accommodate several key optimizations like the ones of [12] without needing to extend the underlying WAM.

The rest of the paper proceeds as follows. In Sec. 2 we present the architecture of the library and the interface of the modules. In Sec. 3 we discuss with an example how to use the glass box approach at different levels for better efficiency in a particular problem, with preliminary benchmarks illustrating the gains. Finally, in Sec. 4 we conclude and discuss related and future work.

2 Architecture of the Ciao CLP($\mathcal{FD}$) Library

The Ciao CLP($\mathcal{FD}$) library consists of seven modules grouped into three logical layers plus two specialized Prolog to Prolog translators. In the definition of these modules and interfaces we profit from Ciao's module system [13] and Ciao's support for assertions [14,4], so that every predicate is correctly annotated with its types and other relevant interface-related characteristics, as well as documentation. The translators are built using the Ciao *packages* mechanism [13], which provides integrated and modular support for syntax modification and code transformations. A description of the user interface for the library along with up-to-date documentation may be found in the relevant part of the Ciao manual.

2.1 The Global Architecture

The global architecture is illustrated in Fig. 1. The kernel layer provides facilities for range handling and propagation chains, which are used for defining finite domain variables — which, as mentioned before, are different from the standard logical variables. The $\mathcal{FD}$ layer defines a finite set of constraints such as `a+b=c/3`, using indexicals. These constraints are translated form their indexical form to a set of instructions of the kernel layer. Labeling and branch-and-bound optimization search modules complete the finite domain solver.

The CLP($\mathcal{FD}$) constraints are translated to $\mathcal{FD}$ constraints by a CLP($\mathcal{FD}$) compiler. We use attributed variables to attach a finite domain variable to every logical variable involved in CLP($\mathcal{FD}$) constraints. Thus, the CLP($\mathcal{FD}$) layer is thin and of very low overhead.

2.2 The Finite Domain Kernel

The finite domain kernel is the most important part of the library. Its implementation freely follows the design of the GNU Prolog $\mathcal{FD}$ solver ([8] provides a general overview of this solver). A finite domain variable is composed of a range and several propagation chains. When the submission of a constraint modifies the range of a finite domain variable, other finite domain variables depending on that range are updated by firing up constraints stored in propagation chains. The propagation events are executed in a synchronous way, meaning that a range change will fail if any of its dependent constraints cannot be satisfied.
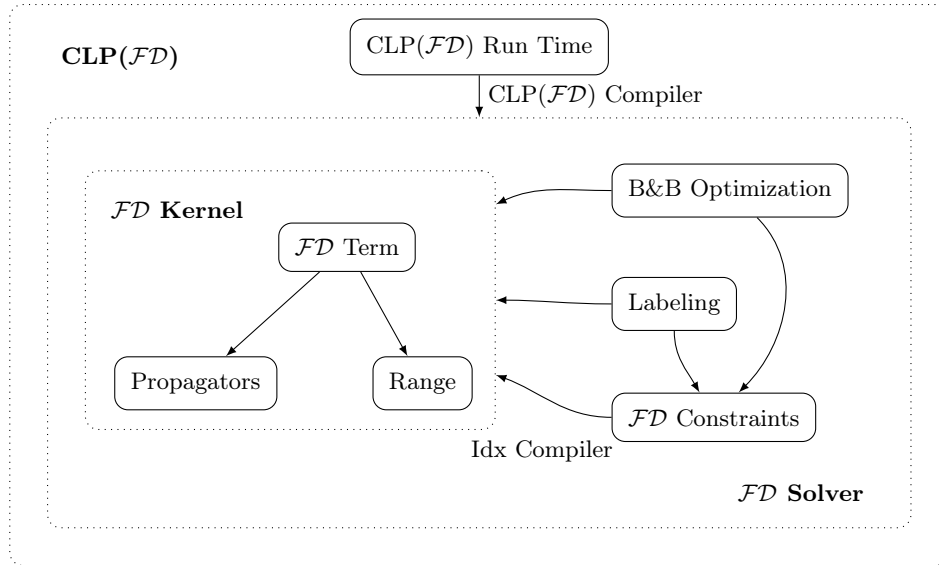
Fig. 1. The Ciao CLP($\mathcal{FD}$) Library Architecture.

The kernel implements arithmetic over ranges (pointwise operations, union, intersection complementation, ...) and management of propagation chains, amounting to the delay of Prolog goals on arbitrary events. These two elements are used to implement the two basic operations of a finite domain variable: `tell` and `prune`. The first one attempts to constrain a variable into a particular range, while the second one (`prune/2`) removes a value form the range of a variable. The variable code inspects the new and old ranges and wakes up the suspended goals on a given variable.

All the data structures are coded in an object-oriented style. Efficient access and in-place update are implemented by using the `setarg/3` primitive. We took special care to use `setarg/3` in a *safe* way to avoid undesired side effects, such as those described by Tarau [15].

Ranges. Range handling is one of the most important parts of the library, given the high frequency of range operations. Indeed, the library supports three implementations for ranges: the standard one using lists of closed integer intervals; an implementation using lists of open (i.e., unbounded) intervals; and a bit-based implementation which despite allowing unbound ranges is more suitable for problems dealing with small ranges.[3] Indeed, the user is encouraged to implement new range modules which are better suited to some particular problems.

---

[3] The implementation of the bit-based range uses arbitrary precision integers plus three non-ISO predicates for computing the least and most significative bits, and the number of active bits in such integers. We implemented these predicates in C.

```
r ::=   t .. t                 (interval)
      | {t}                    (singleton)
      | r \/ r                 (union)
      | r /\ r                 (interseton)
      | - r                    (complementation)
      | r + n                  (pointwise addition)
      | r - n                  (pointwise subtraction)
      | r * n                  (pointwise multiplication)

t ::=   min(Y)                 (minimum)
      | max(Y)                 (maximum)
      | dom(Y)                 (domain)
      | val(Y)                 (value)
      | t+t | t-t | t*t | ...  (arithmetic expression)
      | n                      (bound)
```

Fig. 2. Range Interface, Part 1: Syntax.

The interface that a range module must implement is split into two parts. The first one, shown in Fig. 2, deals with range creation and manipulation. Each of the operations defined in the figure has a corresponding predicate. For instance, bounds addition `t+t` is implemented by the predicate `bound_add/3`, and similarly for the rest of the predicates. Note that it is a convention of the interface that any operation that tries to create an empty range will fail. This is better for efficiency and we found no practical example yet where this would be inconvenient.

Fig. 3 lists the rest of the predicates that a range implementation must provide. They are mainly used for obtaining information about a range and are instrumental for the labeling algorithms.

Propagation Chains. Propagation chains are just lists of goals meant to be executed when a change in the range of a $\mathcal{FD}$ variable happens. The module defines a *propagation chain structure* that is simply a named set of chains. We support in-place update for the structure, thus allowing efficient update of the propagation chains used in the finite domain variables. The interface of the

`fd_range_bound_t/1`	Type of a range bound.
`fd_range_t/1`	Type of a range object.
`is_singleton/1`	True if range is a singleton.
`singleton_to_bound/2`	Returns the value of a singleton range.
`size/2`	Number of elements in a range.
`get_domain/2`	List of elements in a range.
`enum/2`	Backtracks throughout all the elements in a range.
`bound_const/2`	Correspondence of indexical constants with bounds.

Fig. 3. Range Interface, Part 2: Predicates.

`fd_pchains_t/1`	Type of a chain structure.
`fd_pchain_type_t/1`	Name of a chain.
`empty/1`	Returns an empty chain structure.
`add/3`	Adds a goal to a given chain.
`execute/2`	Wakes up a particular chain.

Fig. 4. Propagation Chain Interface.

propagation chain module is presented in Fig. 4. We use internal facilities of the Ciao module system in order to efficiently implement `execute/2`.

Finite Domain Variables. An $\mathcal{FD}$ variable is a structure consisting of a range and a propagation chain.

In the current implementation, integers are considered to be finite domain variables too. However, we are in the process of phasing out this optimization as we incorporate more information into finite domain variables to aid optimizations.

$\mathcal{FD}$ variables are never unified, i.e., they cannot be substituted by others or by integer values as is typically done by the Prolog unification mechanism. A priori, such variables have no correspondence to Prolog logical variables.

Apart from accessing its range and propagation chain, the most important operations that a finite domain variable supports is the tell operation, which tries to update the $\mathcal{FD}$ variable to a new range:

```
1  tell_range(FdVar, TellRange):-
2          fd_var:get_range(FdVar, VarRange),
3          fd_range:intersect(VarRange, TellRange, NewRange),
4          set_range_and_propagate(FdVar, VarRange, NewRange)
```

The propagation predicate will set the new range for the variable and compare the new range with the old one. The current definition — following [12] — supports four propagation events, depending on the range change:

dom: The range changed.
max: The maximum of the range has changed.
min: The minimum of the range has changed.
val: The new range is a singleton.

2.3 The Finite Domain Solver

Once the finite domain kernel is in place, the finite domain solver is just the labeling algorithm and a set of constraints defined using the kernel. As mentioned before, the constraints are defined using indexicals, of the form `X in Range`. Such indexicals are compiled to programs of the $\mathcal{FD}$ kernel in a transparent way for the user. The compilation is carried out by Ciao's source-to-source transformation capabilities, which means that an input Prolog file using the `indexicals` package

is processed in such a way that predicates containing indexical definitions are replaced by their compiled form.

The indexical syntax is intended to be compatible with syntax used in SIC-Stus and GNU Prolog. However, the use of Ciao's package system means that the user may freely mix indexicals with Prolog code (or with many other syntax extensions, such as, e.g., functional notation) without any ill effect, as seen in Appendix A.

The Constraints Library. A reasonable set of local constraints is provided, covering most examples that we have tried to date. We use the convention of using `t` for ground terms, such that in the constraint `'a+b<>c'/3`, all three arguments are assumed to be $\mathcal{FD}$ variables, whereas in the constraint named `a+t<>c/3`, the second argument is assumed to be an immutable singleton, and thus no propagation chains will be installed on it.

Labeling and Optimization Searches, This layer includes also typical labeling algorithms and branch and bound optimization searches. In fact, the current labeling engine is a slight adaptation of the one in the SWI CLP($\mathcal{FD}$) library: we opted for replacing the preliminary version of the engine with this one from SWI, because of its many useful features and easy adaptability to our library.[4] The porting task was relatively easy because the labeling engine is a quite peripheral part of the library (i.e., it has very few code dependencies). It also underlines the high modularity of our library, since two versions of the labeling are in fact available [5]. Finally we obtained for free a common user interface with SWI (and Yap).

The optimization searches uses a branch-and-bound algorithm with restart to find a value that minimizes (or maximizes) the $\mathcal{FD}$ variable according the execution of a Prolog goal. It offers a user-interface similar to the one provided by GNU Prolog.

2.4 CLP($\mathcal{FD}$)

With the $\mathcal{FD}$ solver in place, supporting the CLP($\mathcal{FD}$) paradigm is a matter of performing two mappings: logical variables must be put in correspondence with $\mathcal{FD}$ variables and CLP($\mathcal{FD}$) constraints must be translated to $\mathcal{FD}$ constraints.

Variable Wrapping. For every logical variable to be involved in a CLP($\mathcal{FD}$) constraint we will attach to it an attribute containing an $\mathcal{FD}$ variable:

---

[4] Some features of this engine are currently disabled, but we are planning to activate all such features shortly. The labeling engine was in fact extracted from the *tor* library [16], where it is isolated in a single file.

[5] The old labeling engine can be found in revisions older than 14721 of Ciao 1.15.

```
1   wrapper(A, X):- get_attr_local(A, X), !.
2   wrapper(A, X):- var(A), !, fd_term:new(X), put_attr_local(A, X).
3   wrapper(X, X):- integer(X), !.
```

Logical variables and finite domain variables may communicate in two ways. In
the first one, two logical variables may be unified, needing to link their underlying
finite domain variables. We implement this communication using the `unify_hook`
attribute:

```
1   attr_unify_hook(IdxVar, Other):-
2         ( nonvar(Other) ->
3            ( integer(Other) ->
4                fd_constraints:'a=t'(IdxVar, Other)
5            ; clpfd_error(type_error(Other), '='/2)
6            )
7         ; get_attr_local(Other, IdxVar_) ->
8              fd_constraints:'a=b'(IdxVar, IdxVar_)
9         ; put_attr_local(Other, IdxVar)
10        ).
```

We simply call the $\mathcal{FD}$ constraints `'a=b'/2` and `'a=t'/2`.

The other form of communication is instantiation of the logical variable
when the corresponding finite domain one gets a singleton range. We modify
the `wrapper` predicate to add an instantiation goal to the val chain of freshly
created $\mathcal{FD}$ vars, i.e., we replace the second clause within the definition of the
wrapper by the following one:

```
1   wrapper(A, X):- var(A), !, fd_term:new(X), put_attr_local(A, X),
2         % Force instantiation of A when X represents an integer
3         fd_term:add_propag(X, val, 'fd_term:integerize'(X, A)).
```

This small example points out the possibilities of our scheme beyond the current
use as a support for indexicals.

Constraint Compilation. The $\mathcal{FD}$ solver provides a finite set of $\mathcal{FD}$ con-
straints, however, in the CLP($\mathcal{FD}$) side we may encounter constraints such as:

```
1       A #= B + C + D + E
```

which should be linearized to

```
1       A1 #= D + E,
2       B1 #= B + C,
3       A #= A1 + B1
```

and then wrapped to:[6]

---

[6] We profit here from Ciao's functional notation such that for `p(X,Y)`, `~p(X)` is han-
dled syntactically like a function with return value `Y`.

```
1        'a=b+c'(~wrapper(A1), ~wrapper(D), ~wrapper(E)),
2        'a=b+c'(~wrapper(B1), ~wrapper(B), ~wrapper(C)),
3        'a=b+c'(~wrapper(A) , ~wrapper(A1), ~wrapper(B1))
```

We provide a small compiler which takes care of this process, along with other features like compile-time integer detection.

3   Glass-Box Programming

As previously stated, we encourage the use of a glass box approach when programming with this library. We will use the classical queens program in order to illustrate some of the possibilities that the library offers:

- The use of different range implementations.
- The direct use of the $\mathcal{FD}$ constraints, skipping the CLP($\mathcal{FD}$) compiler.
- The definition of new $\mathcal{FD}$ constraints using indexicals.
- The definition of new atomic constraints directly using the solver kernel, thus skipping the indexical compiler.

Benchmarking Conditions: We provide for illustration purposes some preliminary experimental results. However, it is important to point out that the library is not yet in a state in which relevant absolute performance numbers can be produced and its performance potential fully assessed, since it is still missing important optimizations. Also, only two benchmarks are used.

The benchmarks were run using Ciao 1.15 (revision 14744) on an Intel(R) Core(TM)2 CPU T7200 @ 2.00GHz computer. For reference, we include also the corresponding numbers for SWI Prolog (v. 5.10.4). The purpose is not to make an extensive comparison[7] but rather to have a simple, well understood baseline with which to compare. We should note that we did not explore SWI's support for custom constraints. At the same time, during these tests we have determined that backtracking over changes made by `setarg/3` is currently significantly slower in Ciao than in SWI, which, given the reliance of the implementation on `setarg/3` gives us a clear avenue for performance improvement, independently of any changes to the library itself.

The complete program used in the benchmark is shown in Appendix A. Basically a benchmark has three run time parameters, the number of queens (n=N), the labeling strategy (either "step" or "first fail"[8]), and the constraints used, whose meaning will be explained later. For SWI, only the first two parameters carry significance.

---

[7] This is left as future work where, in addition to implementing the optimizations mentioned, we will include comparison with a number of other systems as well.

[8] Comparing the Ciao and SWI libraries using the heuristic labeling strategies as "first fail" is relevant since both use the same code for labeling.

Queens Parameters	Bits	Closed	Open	SWI
n=16, step, clpfd	0.916	1.144	1.432	1.050
n=16, step, fd	0.572	0.848	1.104	–
n=16, step, idx	0.388	0.648	0.916	–
n=16, step, kernel	0.224	0.336	0.368	–
n=90, ff, clpfd	2.080	2.052	2.484	1.071
n=90, ff, fd	1.112	1.272	1.592	–
n=90, ff, idx	0.752	1.124	1.588	–
n=90, ff, kernel	0.388	0.408	0.432	–

Fig. 5. Queens Benchmark.

3.1 Range Implementations

As previously stated, the library provides three range implementations, selectable at compile-time. The standard one is called "Closed," and represents ranges using a Prolog list of intervals of integers. Thus, every $\mathcal{FD}$ variable is always bound. "Open" is a variation of this approach where the intervals are enriched with constants `sup` and `inf`. This imposes a penalty on bound arithmetic. Lastly, we compare both against a simple bit-vector implementation, done mostly in Prolog with a small support from C. The results can be seen in Fig. 5. The differences go from negligible to more than 50%. In a different benchmark (bridge), the closed interval version was 25% faster than the open one.

3.2 Constraint Implementations

We now focus on the different possibilities that the library allows for $\mathcal{FD}$ constraint programming.

In the queens program, the main constraint of the problem is expressed by the `diff/3` constraint:

```
1  diff(X, Y, I) :-
2      X #\= Y,
3      X #\= Y+I,
4      X+I #\= Y.
```

where `I` will be always an integer.

However, the compiler cannot (yet) detect that `I` is an integer, and may perform some unnecessary linearization. We may skip the compiler and define `diff` using directly the $\mathcal{FD}$ constraints:

```
1  diff(X, Y, I):-
2      fd_constraints:'a<>b'(~w(X),~w(Y)),
3      fd_constraints:'a<>b+t'(X, Y, I),
4      fd_constraints:'a<>b+t'(Y, X, I).
```

The speedup is considerable, getting close to 50% speedup in some cases. Indeed, the compiler should be improved to produce this kind of code by default.

The user may notice that the above three constraints may be encoded by using just two indexicals. For instance one can use the following definition for `diff/3`:

```
1  diff(X,Y,I):-
2      idx_diff(~w(X), ~w(Y), I).
3  idx_diff(X, Y, I) +:
4      X in -{val(Y), val(Y)+c(I), val(Y)-c(I)},
5      Y in -{val(X), val(X)+c(I), val(X)-c(I)}.
```

Again, the improvement is up to 40% from the previous version.

However, the constraint `diff` can be improved significantly by using directly the kernel delay mechanism (val chain) and $\mathcal{FD}$ variable operations. In particular, we use the optimized kernel `prune/2` operation that removes a single element form the range of a variable:

```
1   diff(X, Y, I):-
2          wrapper(X, X0), wrapper(Y, Y0)
3          fd_term:add_propag(Y, val, 'queens:cstr'(X0, Y0, I)),
4          fd_term:add_propag(X, val, 'queens:cstr'(Y0, X0, I)).
5
6   % Y is always a singleton.
7   cstr(X, Y, I):-
8          fd_term:integerize(Y, Y0),
9          fd_term:prune(X, Y0),
10         Y1 is Y0 + I,
11         fd_term:prune(X, Y1),
12         Y2 is Y0 - I,
13         fd_term:prune(X, Y2).
```

We reach around 80% speedup from the first version, and this result is optimal regarding what the user can do. Additional speedups can be achieved, but not without going beyond our glass-box approach. Indeed, our CLP($\mathcal{FD}$) compiler is simpler given that we are working on a new translator that directly generates custom kernel constraints from CLP($\mathcal{FD}$) constraints.

4   Conclusions and Future Work

The Ciao CLP($\mathcal{FD}$) library described is distributed with the latest Ciao version, available at `http://ciaohome.org`. Although included in the main distribution, it lives in the `contrib` directory, as it should be considered at a beta stage of development.

Even if we did not include yet important optimizations that should improve significantly the performance of the library, the current results are encouraging. The library has been used successfully internally within the Ciao development team in a number of projects.

The modular design and low coupling of components allow their easy replacement and improvement. Indeed, every individual piece may be used in a glass-box fashion. We expect that the use of Prolog will allow the integration with Ciao's powerful static analyzers. At the same time, the clear separation of run-time and compile-time phases allows the modification and the improvement of the translation schemes in an independent manner. Indeed, the advantages of this design have already been showcased in [17], where a Prolog to Javascript cross-compiler was used to provide a JS version of the library and which only required replacing a few lines of code. Using this cross-compiler CLP($\mathcal{FD}$) programs can be run on the server side or on the browser side unchanged.

Regarding future work, we distinguish two main lines: the kernel and the CLP($\mathcal{FD}$) compiler.

For the kernel, the first priority is to finish settling down its interface. While we consider it mature, some optimizations — like avoiding reexecution — may require that we include more information in our $\mathcal{FD}$ variable structure, range modification times, etc. Indeed, we would like to support more strategies for propagators than the current linear one. Support for some global constraints is on the roadmap, and will likely mean the addition of more propagation chains.

The library features primitive but very useful statistics. However we think it is not enough and we are working on an $\mathcal{FD}$ instrumentation package that will provide detailed statistics and profiling. This is key in order to extract the maximum performance from the library. Once we get detailed profiling information from a wide variety of benchmarks, a better range implementation will be due.

Regarding the CLP($\mathcal{FD}$) compiler, the current version should be considered a proof of concept. Indeed, we are studying alternative strategies including the generation of custom kernels or specialized $\mathcal{FD}$ constraints for each particular program in contrast to the current approach of mapping a CLP($\mathcal{FD}$) program to a fixed set of primitive constraints. CiaoPP — Ciao's powerful abstract interpretation engine — could be used in the translation, providing information about the CLP($\mathcal{FD}$) program to the CLP($\mathcal{FD}$) compiler so it can generate an optimal kernel of $\mathcal{FD}$ code for that program. In this sense, we think that we will follow the CiaoPP approach of combining inference with user-provided annotations in the new CLP($\mathcal{FD}$) compiler.

Acknowledgments

References

1. Jaffar, J., Maher, M.: Constraint LP: A Survey. JLP **19/20** (1994) 503–581
2. Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation and evaluation of the constraint language cc(fd). Journal of Logic Programming **37**(1–3) (1998) 139–164
3. Dib, M., Abdallah, R., Caminada, A.: Arc-consistency in constraint satisfaction problems: A survey. In: Second International Conference on Computational Intelligence, Modelling and Simulation. (2010) 291–296
4. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming **12**(1–2) (January 2012) 219–252 http://arxiv.org/abs/1102.5497.
5. Dincbas, M., Hentenryck, P.V., Simonis, H., Aggoun, A.: The Constraint Logic Programming Language CHIP. In: Proceedings of the 2nd International Conference on Fifth Generation Computer Systems. (1988) 249–264
6. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Proceedings of the9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Including a Special Trach on Declarative Programming Languages in Education. PLILP '97, London, UK, UK, Springer-Verlag (1997) 191–206
7. D. Diaz, S.A., Codognet, P.: On the implementation of GNU Prolog. Theory and Practice of Logic Programming **12**(1–2) (January 2012) 253–282
8. Codognet, P., Diaz, D.: Compiling constraints in clp(fd). J. Log. Program. **27**(3) (1996) 185–226
9. Zhou, N.F.: Programming finite-domain constraint propagators in action rules. Theory Pract. Log. Program. **6**(5) (September 2006) 483–507
10. Triska, M.: The finite domain constraint solver of swi-prolog. In Schrijvers, T., Thiemann, P., eds.: Functional and Logic Programming. Volume 7294 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2012) 307–316
11. Morales, J.F., Hermenegildo, M.V., Haemmerlé, R.: Modular Extensions for Modular (Logic) Languages. In: 21th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'11), Odense, Denmark (July 2011) To appear.
12. Díaz, D., Codognet, P.: A Minimal Extension of the WAM for `clp(fd)`. In: Proceedings of the Tenth International Conference on Logic Programming, Budapest, MIT press (June 1993) 774–790
13. Cabeza, D., Hermenegildo, M.: A New Module System for Prolog. In: International Conference on Computational Logic, CL2000. Number 1861 in LNAI, Springer-Verlag (July 2000) 131–148
14. Hermenegildo, M., Puebla, G., Bueno, F., López-García, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming **58**(1–2) (2005) 115–140
15. Tarau, P.: BinProlog 2006 version 11.x Professional Edition User Guide. BinNet Corporation. (2006) Available from `http://www.binnetcorp.com/`.
16. Schrijvers, T., Triska, M., Demoen, B.: Tor: Extensible Search with Hookable Disjunction. Draft. Available from `http://users.ugent.be/~tschrijv/tor/` (2012)
17. Morales, J.F., Haemmerlé, R., Carro, M., Hermenegildo, M.V.: Lightweight compilation of (C)LP to JavaScript. Theory and Practice of Logic Programming, 28th Int'l. Conference on Logic Programming (ICLP'12) Special Issue (2012) To appear.

A   Complete Code for the Queens Example

```
1   queens(N, L, Lab, Const) :-
2         length(L, N),
3         domain(L, 1, N),
4         safe(L, Const),
5         labeling(Lab, L).
6
7   safe([], _Const).
8   safe([X|L], Const) :-
9         noattack(L, X, 1, Const),
10        safe(L, Const).
11
12  noattack([], _, _, _Const).
13  noattack([Y|L], X, I, Const) :-
14        diff(Const, X, Y, I),
15        I1 is I + 1,
16        noattack(L, X, I1, Const).
17
18  diff(clpfd, X, Y, I) :-
19        X #\= Y, X #\= Y+I, X+I #\= Y.
20
21  diff(fd, X,Y,I):-
22        fd_diff(~wrapper(X), ~wrapper(Y), I).
23
24  fd_diff(X, Y, I):-
25        fd_constraints:'a<>b'(X,Y),
26        fd_constraints:'a<>b+t'(X,Y,I),
27        fd_constraints:'a<>b+t'(Y,X,I).
28
29  diff(idx, X,Y,I):-
30        idx_diff(~wrapper(X), ~wrapper(Y), I).
31
32  idx_diff(X, Y, I) +:
33        X in -{val(Y), val(Y)+c(I), val(Y)-c(I)},
34        Y in -{val(X), val(X)-c(I), val(X)+c(I)}.
35
36  diff(kernel, X,Y,I):-
37        kernel_diff(~wrapper(X), ~wrapper(Y), I).
38
39  kernel_diff(X, Y, I) :-
40        fd_term:add_propag(Y, val, 'queens:cstr'(X, Y, I)),
41        fd_term:add_propag(X, val, 'queens:cstr'(Y, X, I)).
42
43  cstr(X, Y, I):-
44        fd_term:integerize(Y, Y0),
45        fd_term:prune(X, Y0),
46        Y1 is Y0 + I, fd_term:prune(X, Y1),
47        Y2 is Y0 - I, fd_term:prune(X, Y2).
```